Constructing Transition Models of AI Planner Behavior *

Adele E. Howe Larry D. Pyeatt
Computer Science Department
Colorado State University
Fort Collins, CO 80523
email: {howe,pyeatt}@cs.colostate.edu
URL: http://www.cs.colostate.edu/~{howe,pyeatt}

Abstract

Evaluation and debugging of AI systems require coherent views of program performance and behavior. We have developed a family of methods, called Dependency Detection*, for analyzing execution traces for small patterns. Unfortunately, these methods provide only a local view of program behavior. The approach described in this paper integrates two methods, dependency detection [10] and CHAID-based analysis [3], to produce an abstract model of system behavior: a transition diagram of merged states. We present the algorithm and demonstrate it on synthetic examples and data from two AI planning and control systems. The models produced by the algorithm summarize sequences and cycles evident in the synthesized models and highlight some key aspects of behavior in the two systems. We conclude by identifying some of the inadequacies of the current algorithm and suggesting enhancements.*

1 Introduction

AI programs can behave unpredictably. They often operate in novel domains, may implement new methods/algorithms, and often rely on knowledge or occasional randomness in their choice of action. Often, the domains themselves are not predictable, as in many planning and control applications. From a researcher or developer's viewpoint, these characteristics can make evaluation and debugging a nightmare.

Debugging focuses on local decision making, evaluation on the end result of many decisions. The problem is that local decisions that seem appropriate may add up to overall unsatisfactory performance. AI developers need a technique for characterizing behavior at a higher level than debugging by stepping through execution or poring over reams of execution data, but not as coarse as the summary numbers typically collected for evaluation.

To fill the gap, we have developed a set of statistics based methods for characterizing execution time planner behavior called *Dependency Detection* (DD). In this paper, we describe the most recent: a method for constructing event-based transition models of program behavior from execution traces. This method integrates our basic Dependency Detection method with a Chi Automatic Interaction Detection (CHAID) based method of constructing n-step transition matrices.

DD models the relationships between substrings of discrete events in execution traces of a planner [10]. DD tests whether the occurrence of one event depends on the occurrence of other events prior to it (i.e., whether one set of events appears to cause another event). Extensions to the basic model have been used to find temporally disparate dependencies as well as to compare similar dependencies to determine which exerts the strongest effect.

The CHAID-based analysis constructs transition matrices for describing event traces from distributed systems [3]. The transition matrices include transition probabilities from one event to another in the traces and can be used as the basis for Semi-Markov models.

Although neither DD nor CHAID-based analysis on their own provides a full view of behavior over time, the two methods complement each other in determining aspects of a behavioral model. DD returns snapshots of interactions (sequences); CHAID provides disjoint interaction information, estimated transition probabilities between pairs of events. The approach in this paper integrates the two to produce a model of system behavior that includes both sequences and transitions. The model creates a "big picture" of system behavior in its environment and summarizes the observations

*This research was supported in part by ARPA-AFOSR contract F30602-93-C-0100 and by NSF Research Initiation Award #RIA IRI-930857. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein. We wish to also thank the anonymous reviewers for their careful, gracious and insightful comments.

of actual system state. This paper describes the two approaches and their integration and demonstrates the performance of the method on synthetic data as well as data from two AI systems.

2 Background

The technique described in this paper was inspired by evaluation and debugging efforts in AI. Within AI, evaluation research focuses on metrics for performance and methods for designing and analyzing experiments (see [5] for an overview of this field). Researchers in validation and verification of knowledge based systems have developed methods for measuring and assessing performance in predominantly rule-based systems (see [8] for the most recent proceedings from a continuing workshop series on the topic). Debugging of AI systems has been largely ignored, but two notable exceptions are Chien's research in static and dynamic analysis of planner knowledge bases [4] and Fickas et al.'s work on predicting failure [7].

The difficulty of evaluating and debugging in distributed systems has led to many tools and techniques for visualizing and assessing their performance. While a review of those efforts is well beyond the scope of this paper, one tool that was obviously applicable to the problem of characterizing AI planner behavior is Chitra [13]. The Chitra family of systems (Chitra 92-96) support the modeling and visualization of performance data from parallel and distributed systems [1, 2]. At an abstract level, an AI planner operating in its environment can be characterized as two partially predictable interacting processes.

Our method combines two approaches: one from statistical evaluation of AI systems and one from performance analysis of distributed systems. The remainder of this section will describe the two approaches, Dependency Detection and the CHAID-based analysis.

2.1 Dependency Detection

The purpose of DD is to find predictors of some event (dependencies). From the perspective of debugging, these predictors may be causes of a down stream failure, early symptoms of it, related problems or even just spuriously correlated events. The dependencies are not intended to constitute a complete causal explanation, but rather a concise description of observed detrimental interactions.

Dependencies are detected statistically by analyzing execution traces. An execution trace is a sequence of observed events, which may refer to occurrences in the environment or within the planning system itself. Some of the events are designated as influences (called *precursors* in DD) and others as the behaviors of interest (called *target* events). For example, one event, T_t, is said to be *dependent* on another, P_a, if T_t is observed to occur more often following P_a than following any other event.

The basic idea is to see whether particular patterns stand out from the whole. Whether a pattern stands out depends on a statistical analysis; what patterns are checked depends on how the traces are searched.

Statistical Basis The statistical analysis requires two steps. Combinations of influencing events are first tested for whether they are more or less likely to be followed by each of the target events. Then the significant combinations are compared to remove overlapping combinations. The output of the process is a list of sequences (e.g., $AB \rightarrow C$ which indicates that event C depends on the sequence of events AB) and their "strength" as indicated by the probability that the observed predictor appears as often as other predecessors.

The core of DD is a contingency table test to determine whether a particular target event (e.g., T_t) is more likely after a particular precursor (e.g., P_a). We count four types of sequences in the execution traces: 1) instances of T_t that follow instances of P_a, 2) instances of T_t that follow instances of all precursors other than P_a (abbreviated $P_{\overline{a}}$), 3) target events other than T_t (abbreviated $T_{\overline{t}}$) that follow P_a and 4) target events $T_{\overline{t}}$ that follow $P_{\overline{a}}$. From these frequencies, we construct a two by two contingency table, for example:

	T_t	$T_{\overline{t}}$
P_a	15	10
$P_{\overline{a}}$	30	50

Based on this table, P_a seems to be a good predictor of T_t. A statistical test, the G-test, indicates whether the two events are likely to be dependent [15]; in this case, $G = 5.174, p < .023$, which means that the two events are unlikely to be independent and conclude that T_t depends on P_a. We can use either a G-test or a Chi-square test on the contingency table; we favor the G-test because it supports comparing the dependencies to remove overlaps.

The precursor may be a single event or a combination of events. As a consequence, two precursors may overlap, e.g., $AB \rightarrow C$ and $B \rightarrow C$. A statistical technique based on the G-Test determines which is the best description by comparing the sum of the effects due to the combinations (e.g., $P_X P_a$ for all possible P_Xs) to the effect due to just the grouped effect (e.g., P_a). The intuition behind the test is that if the combinations do not add much information about the effect then they can be disregarded; conversely, if the grouped effect, T_u or P_a, hides differences between the pairs, then the grouped effect should be disregarded as misleading. The output of DD is a list of dependencies that have been found to be significant in the execution traces and that describe the influences without redundancy.

Search for Patterns The basic form of DD searches for all possible length two and three combinations of events found in a set of execution traces. Scaling to longer patterns has two problems: computational and statistical. Computationally, this algorithm requires E^l passes, where E is the number of distinct events and l is the length of the desired pattern. Statistically, a pattern is accepted as a dependency when the probability drops below some threshold; if a large number of tests are conducted, then the likelihood that some subset of the patterns are spurious becomes significant. By introducing heuristics and user guidance into the search, DD has been extended to longer, more complex precursors [9] (e.g., precursors separated by up to eight events, precursors that include wild cards for "don't care" positions, and relative order precursors with an absolute length that can vary over a window) and multiple streams (i.e., events that can be vectors of values) [14].

2.2 CHAID-based Model for Chitra93

The CHAID-based analysis technique was developed for the Chitra93 system for modeling visualizing execution traces from distributed systems [13]. With direction from a user, Chitra93 could develop a continuous time, semi-Markov (CTSM) model, which was based on the Markov-dependence assumption. Thus, the models were limited to representing dependencies between predecessors and their immediate successors. As an enhancement, the CHAID-based model developed for Chitra93 analyzes execution sequences to construct an n-step transition probability matrix [3]. This tool allows a developer to find relationships between pairs of events separated by a set number of other events.

Given a set of target events and a temporal window, the CHAID-based analysis searches for the most predictive position within the window for each of the targets. Informally, CHAID finds dependencies between temporally disconnected events in execution traces. As in dependency detection, CHAID builds a contingency table to determine how likely one set of events is to precede a target event; unlike dependency detection, CHAID includes all events that can appear at that position in the contingency table and so determines the most predictive position, rather than event. An example contingency table for a particular position is:

	A	$\overline{A}$
A	30	171
B	0	674
C	171	313
D	0	325
E	0	313

which is significant ($P < .01$).

1. Collect all subsequences of *window length* and count their frequency in the data,
2. For each *model state*:
   (a) For each independent position within *window length*, construct a contingency table with two columns (model state value and other possible values at the same position) and a row for each possible value of the independent position,
   (b) Find the contingency table with the lowest p-value, which indicates the most predictive independent position,
3. Construct n-step transition matrix that includes the best independent positions where each transition probability is calculated from counts in the contingency tables.

Figure 1. Simplified algorithm for CHAID-based analysis

A simplified form of the algorithm appears in Figure 1. The user directs the algorithm by selecting parameters; for example, *window length* indicates how many positions should be considered as a predictor and *model states* indicates which events to consider as targets. Contingency tables are generated and ranked for all combinations of model states and position. The output is an n-step transition matrix, where the rows are all possible events and the columns are the target events annotated by a position indicator.

As an example of the output, CHAID-based analysis was demonstrated on the Dining Philosopher's problem in which the processes go through one of two sequences, as shown on the left in Figure 2. Sequence one is *TS, A, ES, R*; sequence two is *TL, A, EL, R*. The difference between the sequences is

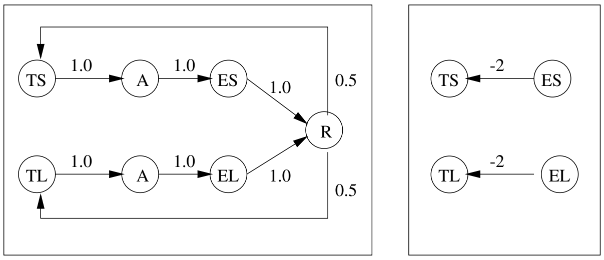


Figure 2. Ideal model (left) and CHAID model (right) for Dining Philosopher's problem. Ideal model labels are transition probabilities; CHAID link labels are steps between events.

that sequences starting with TL include EL later, as opposed to TS and ES. The right side of Figure 2 shows the CHAID-based model generated with window length of 5 and model states of (ES, EL). In this case, the model was generated to find the best position from which to predict the occurrence of states ES and EL. The CHAID-based model selected a delay of two as best predicting their occurrences.

The computational and statistical ramifications of scale up are not as bad as in DD. The complexity of this simplified algorithm is $M * W$ where M is the number of model states and W is the window length.

3 Integrating CHAID and DD to Build Transition Models

Our goal is to develop models that include transitions into and out of states composed of all salient events and that will reconcile single step and temporally disparate transitions. CHAID provides the transition matrix; DD provides subsequences of events over multiple steps.

We are investigating a variety of algorithms that integrate DD and CHAID to develop models. The basic idea is to develop a transition matrix using CHAID and then collapse or expand the state definitions to encompass multiple step/event transitions using DD. The current and simplest version of the algorithm (see Figure 3) refines a 1-step transition matrix from CHAID using a window of length three from DD. We limited the window length in CHAID and the dependency length in DD to minimize the computation time and demonstrate the feasibility of the model construction. A 1-step transition matrix is the minimum required to create a complete model; the dependency length of three (e.g., $AB \rightarrow C$) is the minimum required to connect two transitions from the 1-step transition matrix (e.g., $A \rightarrow B$ and $B \rightarrow C$). Section 6 describes some ideas on how to extend this basic algorithm and ameliorate the current restrictions.

To see how the algorithm works, we can step through a synthetic example. Figure 4 shows an idealized model of some hypothetical program's execution; from this, we generated data that include noise (extra A events added with a probability of 0.1).

For this dataset, CHAID generates the 1-step transition model depicted in Figure 5. While the CHAID generated diagram does capture the essence (e.g., the transitions from A, D and E), it misses the tight coupling of some of the state transitions and is not as concise. In addition, the one step transitions are misleading. For example, the link from B back to itself with a transition probability of 0.24 suggests that a sequence of Bs is likely; however, only a pair of B's is likely, as in the sequence $ABBC$.

For this dataset, DD generates the list of dependencies in Table 1. All of the transitions from the paired states in the ideal model are captured in the dependencies, as well as

1. Read data file of time-stamped events (multiple ensembles/runs can be handled as well),
2. Run CHAID-based analysis and generate 1-step transition matrix,
3. Run DD to generate length 3 dependencies (from which redundancies have been removed),
4. Continue merging states as long as there are states having only one outgoing link:
   - (a) Find a state X with only one outgoing link to Y,
   - (b) Replace state X with state XY,
   - (c) Incoming links to X remain as before,
   - (d) Fix outgoing links for XY:
     - i. Find all dependencies involving XY (e.g., XYZ),
     - ii. Copy transition probabilities for new outgoing links.
   - (e) Fix outgoing links Z_i from original Y:
     - i. Calculate number of occurrences of Y, not including XY,
     - ii. Find dependencies involving YZ_i,
     - iii. Sum transition counts from contingency tables for dependencies not involving X (i.e., not XYZ_i),
     - iv. New probability is occurrences divided by sum.
5. Remove any states with neither incoming or outgoing links.

Figure 3. Algorithm for building unified model

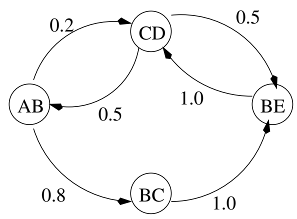


Figure 4. Idealized model (five event model) used to generate execution trace data

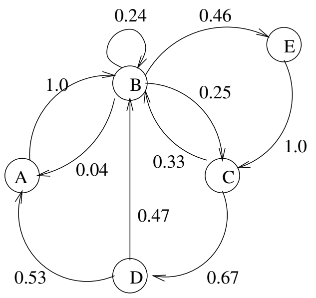


Figure 5. 1-step transition model generated by CHAID from five event model data

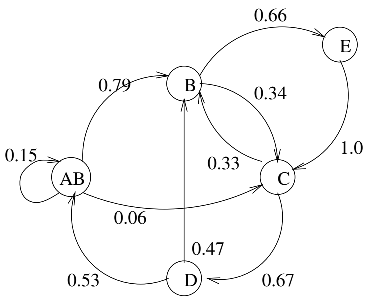


Figure 6. Transition model of Figure 4 data after a single pass to merge states

Dependency	Contingency Table	G	Tran.
$EC \rightarrow D$	(313 0 12 1673)	1903.2	1.0
$BE \rightarrow C$	(313 0 172 1513)	1152.9	1.0
$CD \rightarrow A$	(171 154 30 1643)	931.2	0.53
$CB \rightarrow E$	(159 0 154 1685)	923.0	1.0
$DB \rightarrow E$	(154 0 159 1685)	891.3	1.0
$BB \rightarrow C$	(160 0 325 1513)	538.1	0.
$DA \rightarrow B$	(171 0 503 1324)	364.1	1.0
$BC \rightarrow B$	(159 12 515 1312)	323.7	0.93
$AB \rightarrow B$	(160 42 514 1282)	222.2	0.79
$AB \rightarrow C$	(12 190 473 1323)	57.1	0.06
$AB \rightarrow C$	(12 190 473 1323)	57.1	0.06
$BA \rightarrow B$	(30 0 644 1324)	56.9	1.0
$CD \rightarrow B$	(154 171 520 1153)	37.6	0.47
$BC \rightarrow D$	(12 159 313 1514)	15.2	0.07
$AB \rightarrow A$	(30 172 171 1625)	5.8	0.15

Table 1. Dependencies generated from five event data, ordered by strength. "Tran" is the transition probability for the dependency.

some transitions not in the ideal model (e.g., $BA \rightarrow B$ and $AB \rightarrow A$). Given the 1.0 transitions that can create the sequence $BCBECD$, we would expect to see substrings of the sequence as dependencies; in fact, all of the substrings of length three (e.g., $BC \rightarrow B$, $CB \rightarrow E$, $BE \rightarrow C$, and $EC \rightarrow D$) are dependencies. However, the dependencies themselves form an incomplete picture.

In step four of the algorithm, we use the DD model to reduce the CHAID model and remove some of the inaccurate sequences supported by it. In the first iteration of merging states, we find that A has only one outgoing link to B. So we will overwrite state A with state AB. We cannot delete state B because it has other incoming links (i.e., B, C, and D). The new state's incoming probabilities remain unchanged from the original A, except for its link from B because it has subsumed the original state's incoming links. In step 4d, the new state's outgoing transitions are calculated from the dependencies that match ($AB \rightarrow A$, $AB \rightarrow B$ and $AB \rightarrow C$). The new transition probabilities are calculated by dividing the first entry in each contingency table by the sum of the first entries (e.g., $AB \rightarrow B$ becomes $\frac{160}{202} = 0.79$).

Next (step 4e), the links to and from the original B must be updated to reflect the transfer of some of the state transitions to AB. The outgoing links require some re-calculation because they must reflect paths to B that do not require going through A first. So to calculate those, we find the total number of occurrences of B that were not preceded by A, which in this case is 473 (which can be found by summing the first cell in the contingency table for all the dependencies of the form xBz, where $x \neq A$, which are $BB \rightarrow C$, $CB \rightarrow E$ and $DB \rightarrow E$). Then, we determine the new probabilities for each of the existing links from state B, which are to A, B, C and E. In the cases of outgoing links to A and B, the only dependencies start with AB; consequently, these links have been transferred to the new state AB and become 0 from state B. For the outgoing transition C, we find two dependencies $AB \rightarrow C$ and $BB \rightarrow C$, $AB \rightarrow C$ has been addressed already; the new transition to C becomes 160 (first cell in the contingency table) divided by 473 or 0.34. Finally, by looking up the last dependencies, the transition to E becomes .66. The resulting transition diagram is in Figure 6. While this diagram is similar to Figure 5, note that the link from B back to itself has been removed and that A has been merged into a single state with B.

The above completes the first merging pass using the CHAID transition matrix and the dependency list. We continue iterating through the model merging states until we can no longer find any to merge.

Our current implementation uses the original DD code and the CHAID model algorithm, as it has been implemented here in Lisp. The program accepts a data file in which the lines are events with time-stamps and returns a transition matrix with new probabilities and state names.

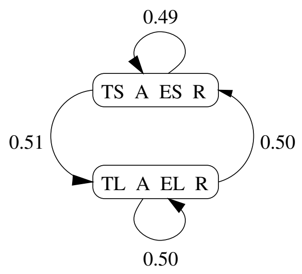


Figure 7. Integrated model for Dining Philosopher's problem

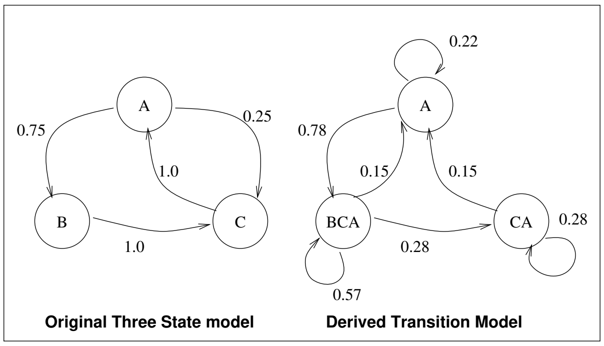


Figure 8. Simple three event model. Original on left, derived on right.

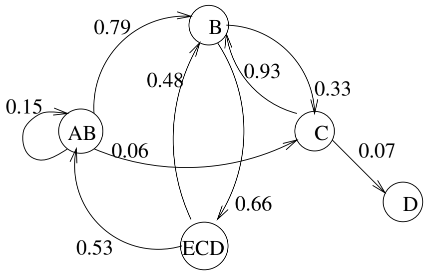


Figure 9. Derived model for five event example.

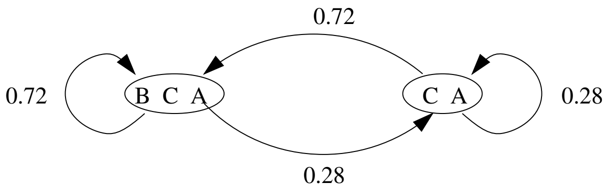


Figure 10. Derived model for three event example, no noise in data.

4 Evaluating Models on Synthetic Data

As a first pass at evaluation, we compared the results of the integrated system to synthesized execution traces. The first example was the Dining Philosopher's as described in section 2.2. We created a data set with no noise, as a base case. The resulting model consisted of two states of sequences and is shown in Figure 7.

For two other examples, the data sets were produced by defining a specific model, then iterating through the model inserting noise (extra states) with some probability. The noise was biased toward repeating the first state in the transition model. The second example was a simple three event model. Figure 8 shows the original and the derived model. The derived model captures the two sequences generated from 1.0 transitions in the ideal model, BCA and CA, and their transitions back to themselves for the possible repetitions. The model also includes A's as a separate state. Unfortunately, this is the noise being captured in the model.

The third example, the five event model, was described in section 3. The final transition model is shown in Figure 9. The algorithm successfully combined the AB and ECD sequences. Many of the probabilities match the original model (e.g., $AB \rightarrow B = .8$ in the original and .79 in the graph and transitions from ECD were roughly evenly divided between AB and B as in the original). The relatively high incidence of extra As were manifest in the transition from AB back to itself. Also, while the graph should be fully connected, there was not enough coherent noise following the few isolated examples of D (not already incorporated in the state ECD) to postulate any transitions out of it. Without sufficient observations in the data, we cannot add a link from D to the graph.

None of the ideal models was exactly replicated by the process. The easiest to replicate are those in which states appear in only one sequence (as in the first example). Hardly surprisingly, merging becomes more difficult as transitions are distributed over more states and more noise is introduced. For example, we re-ran the second example with no noise in the data; the resulting model (see Figure 10) exactly captures the sequences that can be produced by the ideal model. Currently, we are examining how to produce more parsimonious models, but at the expense of completeness.

5 Examples from Two Domains: Phoenix and RARS

From the beginning of the research, the purpose of the modeling has been to expedite evaluation and debugging of planning systems. Planning systems are hard to debug because they tend to include considerable interaction between agents or between a single agent and an only somewhat predictable environment. For a more practical demonstration of our method, we constructed behavioral models for two systems for planning and control: Phoenix and RARS.

5.1 The Phoenix Planner

Dependency Detection was originally motivated by the difficulties encountered in debugging the Phoenix planning system [11]. The Phoenix system encompasses a simulation of forest fire fighting in Yellowstone National Park and the agents that operate within it [6]. Typically, we configured the simulation to include about 10 agents working together over 60-100 hours of simulated time. Decisions about what actions to take often had long term effects that were difficult for programmers to predict.

We designed DD to answer the question: "Does a failure depend on some action or event that preceded it?" with the intention that we would use the information to identify bugs in the planner. We were building a failure recovery component for the planner by incrementally adding capabilities. On the whole, these changes were beneficial; we lowered the overall rate and cost of failure. Unfortunately, not all the effects of the additions were beneficial. For example, at one point, we added two new recovery methods specifically to address two frequent failures and discovered that, as side effect, it became more difficult to fix some other types of failures. We thought it might be due to some long term plan interactions, but were not sure how.

Using DD, we analyzed execution traces from Phoenix and found dependencies between some of the recovery methods and later failures. By searching for static and dynamic connections between the plan actions involved, we identified bugs (e.g., variables calculated incorrectly and improper plan synchronization) using a methodology called *Failure Recovery Analysis*[11].

However, these bugs still involved fairly simple interactions local to just a few events. Some of the more pernicious failures were included in many dependencies, making it difficult to determine what was causing the problem. We used the original execution data[1] on which the recovery code additions were based to generate a transition model.

We expected the model to show a more complicated pattern that could be the cause of the more pernicious failures. Figure 11 shows a fragment of the transition diagram displaying the strongest transitions associated with the `Vip` failure. `Vip` is triggered when the goal of an entire plan is in jeopardy due to combined efforts being inadequate. Three cycles involving `Vip` became apparent and are highlighted in bold. These cycles suggest something akin to a downward spiral: the planner detects a localized failure (`Ccp` and `Ner`) that is not easily repaired (`Rp` is a last ditch recovery method which radically modifies the plan) and leads to the global plan failure (`Vip`). The next step in debugging is to determine whether the available recovery methods for `Ccp` and `Ner` are inadequate or the `Rp` recovery method is flawed. To take that step will require additional state information captured in the execution traces.

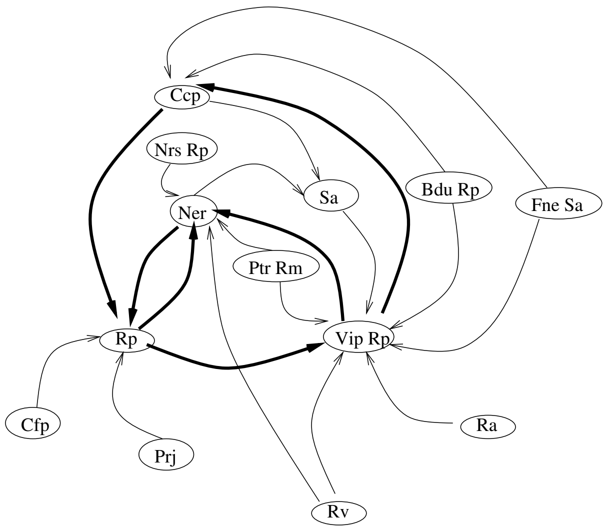


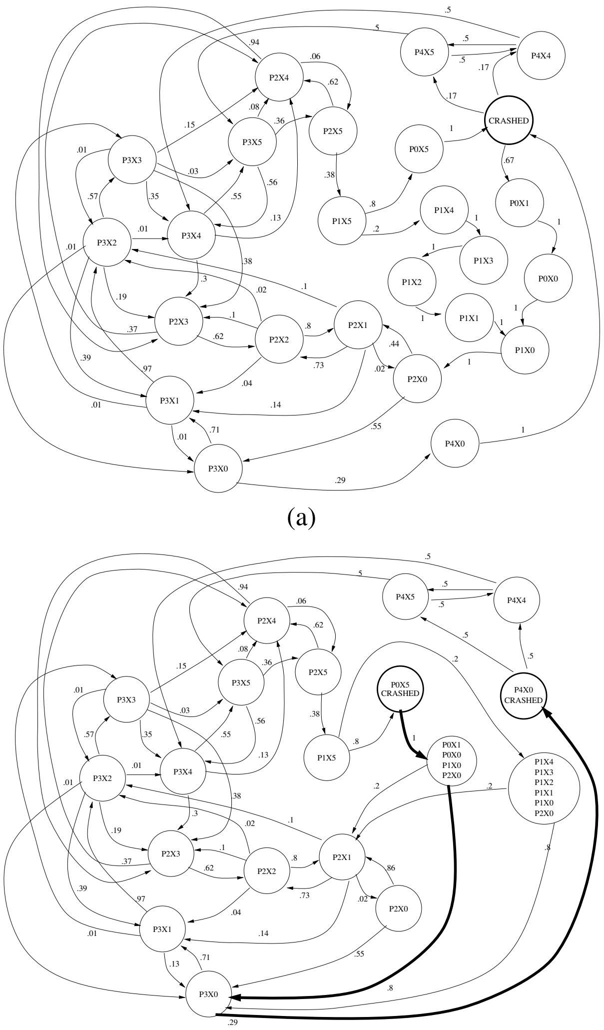
Figure 11. Fragment of derived model for Phoenix example that focuses on the `Vip` failure. Failures are designated by three letter state names and recovery actions by two letter state names.

5.2 The RARS Control System

We are also interested in detecting the causes of agent failures in the Robot Automobile Racing Simulator (RARS) [16]. This simulator is designed to allow different agents to compete in simulated automobile races. RARS runs through the simulation at a fixed lockstep. At each time step, RARS updates the position and velocity of each car based on the environment model and control signals. It then calls the subroutine for each agent in the race to determine their next actions. Each agent routine receives a structure containing information about its position, current velocity, distance to the end of the current track segment, curvature of the track, relative position and velocity of nearby cars, and other information. The agent routine calculates command signals for acceleration and steering. When the agent loses control of the car and runs off of the track (`crash` event), we say that a failure has occurred.

Our current agent simulation uses a quantized subset of the data available to describe its current state. For purposes of generating a model, we included data from two sensors

[1] We had collected almost 500 execution traces (about 120 traces for each of four different failure recovery versions, each representing 60 hours of simulated time). The traces contained 21 different states.



(a)

(b)

Figure 12. Transition models generated from RARS execution data: one step model in (a) and the merged model in (b).

in the state description: track position and speed relative to a wall. Each were prefixed with a letter (P and X, respectively) and quantized into discrete intervals (five for P and six for X). For example, `P0` indicates a position next to the inner wall, and `P4` indicates a position next to the outer wall. `X0` is headed fast toward the outer wall, and `X5` is fast towards the inner wall. We can create an execution trace by collecting, at each time step, the two symbols describing the current state.

Figure 12 shows models extracted from the RARS execution trace; the basic transition model is at the top and the the reduced model is on the bottom. Failure (`crash`) states are highlighted. Many of the states near failure have been merged, giving a better indication of what occurs just before and after a failure occurs. In fact, from the diagram, we can see that states `P1X5` (near the inner wall and headed towards it) and `P3X0` (in the middle and headed towards the outer wall) often lead to failure. Interestingly, sometimes (transition 0.2 from `P1X5`) the agent manages to avoid crashing by changing the speed relative to the wall, thus moving into a sequence of other states that do not easily make the transition to `crash`. On the other hand, we can see by the transition diagram (follow the darkened links) that if it does crash into the inner wall, it often crashes into the outer wall shortly after its recovery. Based on this analysis, our next step is to further investigate the crash recovery and the actions taken at state `P1X5`.

6 What Is Next and What We Learned

Based on the preliminary results, the integration algorithm needs to be extended. Currently, we ignore significant information available from both CHAID and DD. From CHAID, we can determine the most predictive transitions over further periods of time, and from DD, we can determine longer descriptive sequences. We merge only the simplest cases (transitions of 1.0) and prune only the obvious (states without links). Additionally, for any reasonably large number of states, not too surprisingly, the transition diagram becomes difficult to read.

We are extending the basic algorithm in several ways. First, we will use CHAID to determine the best precursor size for DD. Second, we are adding more aggregation and pruning criteria. For example, with aggregation, if CHAID indicates a large step size and the dependencies suggest that the intermediate steps do not matter, we can aggregate new states that include "don't care" events. We can prune state transitions that have little support in the data, i.e., no dependencies or dependencies that are barely significant. Third, we can generate models of variable complexity by modifying the significance threshold for dependencies; the more stringent the threshold, the fewer dependencies will be found and the less support for including transitions. The resulting models will no longer account for all transitions out of states, but simply the most significant and thus less likely to be noise.

We are also investigating methods for displaying the transition diagrams. Several graph drawing packages are publically available. To use these packages, the output needs to translated into the package languages. Because of the complexity of some of the transition diagrams, we may need to augment the translation to display fragments, such as that in Figure 11, that focus on transitions and cycles involving a single state.

As practical applications, we will be incorporating the model generation into two other projects: a debugging assistant and a two layer control system. We have been developing a tool for planner debugging, the Planner Debugging Assistant [12] (PDA). Currently, PDA relies exclusively on dependency detection. With transition models as a supplement, programmers should also be able to determine more complicated sources of bugs (involving more interactions) and possible consequences of changing the code. The second project, which is a followup to the RARS control system, focuses on integrating neural networks for controlling low level behaviors with a symbolic planner for making longer term decisions. One of the key difficulties is keeping the planner apprised of what the neural networks can do. We are exploring using the transition models as a representation that can be generated from the neural network's behavior and interpreted by the planner.

Evaluating model generation is difficult. Ultimately, a model's utility is best measured in terms of how the model aggregation supports system evaluation and debugging. For now, the models summarize the execution traces into a model similar to the originals for the synthesized data and appear to support inference about behavior in the systems: what sequences are inexorable and what patterns of behavior can be observed at an abstract level.

References

[1] M. Abrams, A. Batongbacal, R. Ribler, and D. Vazirani. CHITRA94: A tool to dynamically characterize ensembles of traces for input data modeling and output analysis. Department of Computer Science 94-21, Virginia Polytechnical Institute and State University, June 1994.

[2] M. Abrams and C. Group. Chitra project page. http://info.cs.vt.edu/˜chitra/.

[3] H. T. Cadiz. The development of a CHAID-based model for CHITRA93. Computer science dept., Virginia Polytechnic Institute, February 1994.

[4] S. A. Chien. Static and completion analysis for planning knowledge base development and verification. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS96)*, pages 53–61, Edinburgh, Scotland, 1996.

[5] P. R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[6] P. R. Cohen, M. Greenberg, D. Hart, and A. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, Fall 1989.

[7] S. Fickas, R. Helm, and M. Feather. When things go wrong: Predicting failure in multi-agent systems. Department of Computer and Information Science CIS-TR-91-15, University of Oregon, Eugene, OR, 1991.

[8] R. Gamble and C. Landuaer, editors. *Working Notes of the IJCAI-95 Workshop on Verification and Validation of Knowledge-Based Systems*, Montreal, CA, August 1995.

[9] A. E. Howe. Detecting imperfect patterns in event streams using local search. In D. Fisher and H. Lenz, editors, *Learning from Data: Artificial Intelligence and Statistics V*. Springer-Verlag, 1996.

[10] A. E. Howe and P. R. Cohen. Detecting and explaining dependencies in execution traces. In P. Cheeseman and R. Oldford, editors, *Selecting Models from Data; Artificial Intelligence and Statistics IV*, pages 71–78. Springer-Verlag, NY,NY, 1994.

[11] A. E. Howe and P. R. Cohen. Understanding planner behavior. *Artificial Intelligence*, 76(1-2):125–166, 1995.

[12] A. E. Howe and D. Peterson. Validation and verification in the planner life cycle. In *Working Notes of the IJCAI'95 Workshop on Validation & Verification of Knowledge-Based Systems*, Montreal, CA, August 1995.

[13] T. J. Lee. Chitra93: A tool to analyze system behavior by visualizing and modeling ensembles of traces. Computer science dept., Virginia Polytechnic Institute, April 1994.

[14] T. Oates, D. Gregory, and P. R. Cohen. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifthe International Workshop on Artificial Intelligence and Statistics*, January 1995.

[15] R. R. Sokal and F. J. Rohlf. *Biometry: The Principles and Practice of Statistics in Biological Research*. W.H. Freeman and Co., New York, second edition, 1981.

[16] M. E. Timin. Robot Auto Racing Simulator. Anonymous ftp from `ftp.ijs.com:/rars`, 1995.